Composition

CMSC 202

Code Reuse

- Effective software development relies on reusing existing code.
- Code reuse must be more than just copying code and changing it which is often the case with procedural languages like C.
- The goal with OOP languages is to reuse classes without changing the code within the class — one OOP technique for code reuse is known as ***composition***.

A Simple Database

- Your favorite boss wishes to implement a simple database of Motorcycles they currently have on the dealership's lot. Their application has only a few requirements...

- They would like record the VIN, Color, Date of Arrival, and Date of Sale each Motorcycle that has entered and left the dealership. A report is required that prints all information for each Motorcycle. Motorcycles must be comparable to avoid duplicate entries in the database. For ease of data entry, it must be possible to make a copy of an existing Motorcycle.

- Your contribution to this project is to design and implement a class named Motorcycle that will represent a single Motorcycle.

Designing a Motorcycle Class: Behaviors/Services

- After reading the problem description, the following behaviors/services have been identified for the Motorcycle class.
  - Create a Motorcycle with a VIN, color, date of arrival, and date of sale.
  - Compare two Motorcycle objects to determine if they are identical.
  - Format a string containing all Motorcycle attributes.
  - Create a new Motorcycle which is the copy of an existing Motorcycle.

Designing a Motorcycle Class: Instance Variables

- To support the required services, a simple `Motorcycle` class could contain instance variables representing each of the Motorcycle's required class members.
- These instance variables would all be class types: name of type `String`, and two dates of type `Date`.
- As a first line of defense for privacy and to provide proper encapsulation, each of the instance variables would be declared `private`.

```
public class Motorcycle {
  private String vin;
  private String color;
  private Date arrived; // Date stores Month (String), Day Year (ints)
  private Date sold;    // null means still on the lot
  // ...
}
```

Designing a Motorcycle Class: Constraints

- In order to exist, a Motorcycle must have (at least) a VIN, color, and a Date of arrival.
  - Therefore, it would make no sense to have a no-argument `Motorcycle` class constructor.
- A Motorcycle that has not been sold does not yet have a date of sale.
  - Therefore, the `Motorcycle` class constructor will need to be able to deal with a null value for date of sale (and/or provide a 3 argument constructor).
- A Motorcycle that has been sold must have had an arrival date that preceded its date of sale.
  - Therefore, when both dates are provided to the ***constructor***, they will need to be checked for validity.

Designing a Motorcycle Class: The Class Invariant

- A statement that is always true for every object of the class is called a ***class invariant***.
  - A class invariant can help to define a class in a consistent and organized way
- For the `Motorcycle` class, the following should always be true
  - An object of the class `Motorcycle` has a VIN, color, a date of arrival (which are not `null` values), and if the object has a date of sale, then the date of sale is equal to or later than the date of arrival.
- Checking the `Motorcycle` class confirms that this is true of every object created by a constructor, and all the other methods (e.g., the private method `isValidState`) preserve the truth of this statement.

Class Invariant Summary

- The class invariant is stated as part of the class documentation.
- Error checking in the constructor(s) and setters/mutators insure that the class invariant is not violated.
- Methods of the class which do not change the class's state may assume the class invariant holds.

A Motorcycle Class Constructor

```
/**
 * Class Invariance is that a Motorcycle must have a non-null vin, non-null
 * Color, and a date of arrival that precedes its date of sale (if set).
 *
 * @param vin a non-null string to uniquely identify the Motorcycle
 * @param color a non-null string representing the color
 * @param arrived a non-null date of arrival
 * @param sale a date (this may be null indicating the Motorcycle hasn't yet
 *        been sold)
 */
public Motorcycle(String vin, String color, Date arrived, Date sold) {
    // checks the class invariant
    if (!isValidState(vin, color, arrived, sold)) {
        // program exits if the motorcycle would be created illegally
        System.err.println("Invalid State for Motorcycle Construction");
        System.exit(0);
    }
    this.vin = vin;
    this.color = color;
    this.arrived = arrived;
    this.sold = sold;
}
```

Checking the Class Invariant

```
private boolean isValidState(String vin, String color,
                             Date arrived, Date sold) {
    return vin != null && !vin.equals("") &&
           color != null && !color.equals("") &&
           arrived != null &&
           (sold == null || arrived.before(sold));
}
```

- Which part of the invariance are we validating?

Composition

- Note that the Motorcycle class contains two class type instance variables (String and Dates).

```
private String vin;
private String color;
private Date arrived;
private Date sold;
```

- The use of classes as instance variables is a design method known as ***aggregation*** or ***composition***.
- Composition is a fundamental way to reuse code, but there are coding considerations when composition is used.

Composition Considerations

- With composition, Motorcycle becomes a user of the Date and String classes.
- The Motorcycle class has no special privileges with respect to Date or String.
- The Motorcycle class should delegate responsibility to the Date and String classes whenever possible.
  - We let each class “do the work” of that object.

Designing a Motorcycle's Class: The equals Method

- The definition of `equals` for the class `Motorcycle` includes an invocation of `equals` for the class `String`, and an invocation of the method `equals` for the class `Date`.
- The Motorcycle class passes responsibility for determining equality to the String and Date classes invoking their `equals` methods.
  - This is an important example of code reuse arising from the use of composition to implement Motorcycle.
- Java determines which `equals` method is being invoked from the type of its calling object.

Designing a Motorcycle Class: The equals Method

```
public boolean equals(Motorcycle other) {
    return vin.equals(other.vin) &&
           color.equals(other.color) &&
           arrived.equals(other.arrived) &&
           sold.equals(other.sold);
}
```

These call the ***equals*** method defined in the String class

These call the ***equals*** method defined in the Date class

Warning — Potential Statement of Mass Destruction!

Consider our ***Class Invariant***

- The date of sale could be null.
- Using the equals method could generate a ***null pointer exception***.
- Instead of storing a ***null*** value in sold…
  - We could initialize it with a unique value indicating they are still on dealership's lot (e.g. a date way in the future), or
  - We could add the code first check and see if values are null and then handle accordingly

Designing a Motorcycle Class: The toString Method

- The `Motorcycle` class `toString` method includes invocations of the `Date` class `toString` method.
- Again, an example of code reuse and delegation of responsibility due to composition.

```
public String toString() {
    String state = "";
    state += "Vin: " + vin + "\n";
    state += "Color: " + color + "\n";
    state += "Arrived: " + arrived + "\n";
    state += (sold != null) ? "Sold: " + sold : "Not Sold" + "\n";
    return state;
}
```

Each (non-primitive) instance variable calls toString(), effectively delegating the work to that class

Designing a Motorcycle Class: Making a Copy

- Making a copy of an object requires a special method called a ***copy constructor***.
- A ***copy constructor*** is a constructor with a single argument of the same type as the class.
- The copy constructor should create an object that is a separate, independent object, but with the instance variables set so that it is an exact copy of the argument object.

Copy Constructor for a Class with Primitive Type Instance Variables

```
// a class that does not use composition can
// simply copy the values of the primitive instance
// variables
public Date(Date date) {
    // Not a real date object parameter
    if (date == null) {
        // we'll handle errors differently later
        System.out.println("Fatal Error!");
        System.exit(0);
    }

    // just copy the primitive variables using assignment
    // month is a String which is NOT primitive, but that's ok
    month = date.month;
    day = date.day;
    year = date.year;
}
```

Copy Constructor for a Class Using Composition

- Because of composition, the technique used with Date will not work correctly with Motorcycle in its current form…

```
public Motorcycle(Motorcycle other) {
    if(other == null) {
        System.out.println("Fatal error!");
        System.exit(0);
    }
    vin = other.vin;              // ok
    color = other.vin;            // ok
    arrived = other.arrived;      // dangerous
    sold = other.sold;            // dangerous
}
```

This code would not create an independent copy of the original object. Why not?

Copy Constructor for a Class with Class Type Instance Variables

- The actual copy constructor for the `Motorcycle` class needs to be made "safe."
- Should create completely new and independent copies of `arrived` and `sold`, and therefore, a completely new and independent copy of the original `Motorcycle` object
- For example:

```
arrived = new Date(other.arrived);
```

- Note that in order to define a correct copy constructor for a class that uses composition, copy constructors must already be defined for the instance variables classes (e.g. Date).

Copy Constructor for a Class Using Composition

```
public Motorcycle(Motorcycle other) {
    if(other == null) {
        System.out.println("Fatal error!");
        System.exit(0);
    }
    vin = other.vin;                        // ok
    color = other.vin;                      // ok
    arrived = new Date(other.arrived); // ok
    sold = new Date(other.sold);            // ok
}
```

- Why do we not have to invoke a copy constructor with vin and color?
  - Strings are ***immutable*** objects.
- Why is it necessary to check to see if other == `null` ?
  - A ***null pointer exception*** will occur if other is not an instantiated Motorcycle object.

Using and Misusing References

- When writing a program, it is very important to insure that private instance variables remain truly private.
- For a primitive type instance variable, just adding the `private` modifier to its declaration should insure that there will be no ***privacy leaks***.
- For a class type instance variable, adding the `private` modifier alone is not sufficient.

Pitfall: Privacy Leaks

- The previously illustrated examples from the Motorcycle class show how an incorrect definition of a copy constructor can result in a ***privacy leak***.
- A similar problem can occur with incorrectly defined mutator or accessor methods.

```
public Date getArrivalDate() {
    return arrived;
}
```

Dangerous

```
public Date getArrivalDate() {
    return new Date(arrived);
}
```

Safe

Composition with Arrays

- Just as a class type can be used as an instance variable, arrays can also be used as instance variables.
- We can define an array with a primitive base type.

```
private double[] grades;
```

- Or, an array with a class base type.

```
private Date[] dates;
```

Privacy Leaks with Array Instance Variables

- If an accessor method is provided for the array, special care must be taken just as when an accessor returns a reference to any private object.

```
public double[] getGrades() {
    return grades;
}
```

- The example above will result in a privacy leak.
- Why?

Privacy Leaks with Array Instance Variables

- The previous accessor method would simply return a reference to the array `grades` itself.
- Instead, an accessor method should return a reference to a ***deep copy*** of the private array object.
- Below, `grades` is an array which is an instance variable of the class containing the `getGrades` method.

```
public double[] getGrades() {
    double[] temp = new double[grades.length];
    for (int i = 0; i < grades.length; i++) {
        temp[i] = grades[i];
    }
    return temp;
}
```

Privacy Leaks with Array Instance Variables

- If a private instance variable is an array that has a mutable class as its base type, then copies must be made of each class object in the array when the array is copied.

```
public Date[] getDates() {
    Date[] temp = new Date[dates.length];
    for (int i = 0; i < dates.length; i++) {
        temp[i] = new Date(dates[i]);
    }
    return temp;
}
```

But What If…

- …the user really wants to change the array within the class?
  - The user shouldn't know that the class uses an array.
  - The array must represent some abstract data element in the class (e.g. student grades).
  - Provide a method that changes the abstract data element without revealing the existence of an array.

Remember…

Keep it secret,
keep it safe